11

Tasks and Asynchronous Programming

WHAT'S IN THIS CHAPTER?

- The importance of asynchronous programming
- Using the `async` and `await` keywords with the task-based async pattern
- Creating and using tasks
- Foundations of asynchronous programming
- Error handling with asynchronous methods
- Cancellation of asynchronous methods
- Async streams
- Asynchronous programming with Windows apps

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/Tasks`.

The code for this chapter is divided into the following major examples:

- TaskBasedAsyncPattern
- TaskFoundations
- ErrorHandling
- AsyncStreams
- AsyncDesktopWindowsApp

All the sample projects have nullable reference types enabled.

WHY ASYNCHRONOUS PROGRAMMING IS IMPORTANT

Users find it annoying when an application does not immediately react to requests. As we scroll through a list, we have become accustomed to experiencing a delay because we've learned that behavior over several decades. We are accustomed to this behavior when using the mouse. However, with a touch UI, we often don't accept such a delay. An application with a touch UI needs to react immediately to requests. Otherwise, the user tries to redo the action, possibly by touching the screen more firmly.

Because asynchronous programming was hard to achieve with older versions of .NET, it was not always done when it should have been. One of the applications that blocked the UI thread fairly often is an older version of Visual Studio. With that version, opening a solution containing hundreds of projects meant you could take a long coffee break. Visual Studio 2017 offered the *Lightweight Solution Load* feature, which loads projects only as needed and with the selected project loaded first. Since Visual Studio 2015, the NuGet package manager is no longer implemented as a modal dialog. The new NuGet package manager can load information about packages asynchronously while you do other things at the same time. These are just a few examples of important changes built into Visual Studio related to asynchronous programming.

Many APIs with .NET offer both a synchronous and an asynchronous version. Because the synchronous version of the API was a lot easier to use, it was often used where it wasn't appropriate. With the Windows Runtime (WinRT), if an API call is expected to take longer than 40 milliseconds, only an asynchronous version is available. Since C# 5.0, programming asynchronously is as easy as programming in a synchronous manner, so there shouldn't be any barriers to using the asynchronous APIs, but of course there can be traps, which are covered in this chapter.

C# 8 introduced async streams that make it easy to consume async results continuously. This topic is covered in this chapter as well.

> **NOTE** *.NET offers different patterns for asynchronous programming. .NET 1.0 defined the async pattern. With this pattern,* `BeginXX` *and* `EndXX` *methods are offered. One example is the* `WebRequest` *class in the* `System.Net` *namespace with the* `BeginGetResponse` *and* `EndGetResponse` *methods. This pattern is based on the* `IAsyncResult` *interface and the* `AsyncCallback` *delegate. When using this pattern with the implementation of Windows applications, it is necessary to switch back to the user interface (UI) thread after the result is received.*
>
> *.NET 2.0 introduced the event-based async pattern. With this pattern, an event is used to receive the asynchronous result, and the method to invoke has the Async postfix. An example is the* `WebClient` *class (an abstraction of* `WebRequest`*) with the method* `DownloadStringAsync` *and the corresponding event* `DownloadStringCompleted`*. Using this pattern with Windows applications where a synchronization context is created, it's not necessary to switch to the UI thread manually. This is done from the event.*
>
> *With new applications, you can ignore the methods offered by these patterns. Instead, C# 5 introduced the task-based async pattern. This pattern is based on .NET 4 features, the task parallel library (TPL). With this pattern, an asynchronous method returns a* `Task` *(or other types offering the* `GetAwaiter` *method), and you can use the* `await` *keyword to wait for the result. Methods usually have the Async postfix with this pattern as well. A modern class for doing network requests for implementing this pattern is* `HttpClient` *with the* `GetAsync` *method.*
>
> *Both the* `WebClient` *and* `WebRequest` *classes offer the new pattern as well. To avoid a naming conflict with the older pattern,* `WebClient` *adds* `Task` *to the method name—for example,* `DownloadStringTaskAsync`*.*
>
> *With new clients, just ignore the* `Begin`*/*`End` *methods, and the events based on the async pattern with the classes that offer this functionality to support legacy applications.*

TASK-BASED ASYNC PATTERN

Let's start with using an implementation of the task-based async pattern. The `HttpClient` class (which is explained in more detail in Chapter 19, "Networking") among many other classes implements this pattern. Nearly all methods of this class are named with an `Async` postfix and return a `Task`. This is the declaration of one overload of the `GetAsync` method:

```
public Task<HttpResponseMessage> GetAsync(Uri? requestUri);
```

The sample application uses these namespaces besides the `System` namespace:

```
System.Net.Http

System.Threading.Tasks
```

With the sample application, a command-line argument can be passed to start the application. If a command-line argument is not set, the user is asked to enter a link to a website. After the `HttpClient` is instantiated, the `GetAsync` method is invoked. Using the `await` keyword, the calling thread is not blocked, but the result variable response is only filled as soon as the `Task` returned from the `GetAsync` method is completed (the task status will have the state `RunToCompletion`). When you use the `async` keyword, there's no need to specify an event handler or pass a completion delegate as was necessary with the older async patterns. The `HttpResponseMessage` has a `IsSuccessStatusCode` property that is used to verify if the response from the service was successful. With a successful return, the content is retrieved using the `ReadAsStringAsync` method. This method returns `Task<string>` that can be awaited as well. As soon as the result is available, the first 200 characters of the string HTML are written to the console (code file `TaskBasedAsyncPattern/Program.cs`):

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

string uri = (args.Length >= 1) ? args[0] : string.Empty;
if (string.IsNullOrEmpty(uri))
{
  Console.Write("enter an URL (e.g. https://csharp.christiannagel.com): ");
  uri = Console.ReadLine() ?? throw new InvalidOperationException();
}
using HttpClient httpClient = new();
try
{
  using HttpResponseMessage response = await httpClient.GetAsync(new Uri(uri));
  if (response.IsSuccessStatusCode)
  {
    string html = await response.Content.ReadAsStringAsync();
    Console.WriteLine(html[..200]);
  }
  else
  {
    Console.WriteLine($"Status code: {response.StatusCode}");
  }
}
catch (UriFormatException ex)
{
  Console.WriteLine($"Error parsing the Uri {ex.Message}");
}
```

```
catch (HttpRequestException ex)
{
  Console.WriteLine($"HTTP request exception: {ex.Message}");
}
catch (TaskCanceledException ex)
{
  Console.WriteLine($"Task canceled: {ex.Message}");
}
```

> **NOTE** *The* `using` *declaration that's used with the* `HttpClient` *and the* `HttpResponseMessage` *invokes the* `Dispose` *method at the end of the variable scope. This is explained in detail in Chapter 13, "Managed and Unmanaged Memory."*

To run the program and pass command-line arguments using the .NET CLI, you need to pass two dashes to distinguish the command-line arguments that are meant for the application from the arguments used for the .NET CLI and start the application this way:

```
> dotnet run -- https://csharp.christiannagel.com
```

Using top-level statements, the variable `args` is created automatically. Using `await` with the top-level statements, the generated `Main` method is defined with an `async` scope. When you write a custom `Main` method that uses `await`, it needs to be declared to return a `Task`:

```
public class Program
{
  static async Task Main(string[] args)
  {
    //...
  }
}
```

TASKS

The `async` and `await` keywords are compiler features. The compiler creates code by using functionality from the `Task` class, which you also can write yourself. This section gives information about the `Task` class and what the compiler does with the `async` and `await` keywords. It shows you an effortless way to create an asynchronous method and demonstrates how to invoke multiple asynchronous methods in parallel. You also see how you can change a class to offer the asynchronous pattern with the `async` and `await` keywords.

The sample application uses these namespaces besides the `System` namespace:

```
System.Collections.Generic
System.IO
System.Linq
System.Net
System.Runtime.CompilerServices
System.Threading
System.Threading.Tasks
```

> **NOTE** *This downloadable sample application makes use of command-line arguments, so you can easily verify each scenario. For example, using the .NET CLI, you can pass the* `async` *command-line parameter with this command:* `dotnet run -- -async`. *When using Visual Studio, you can also configure the application arguments in Debug Project Settings.*

To better understand what's going on, the `TraceThreadAndTask` method is created to write thread and task information to the console. `Task.CurrentId` returns the identifier of the task. `Thread.CurrentThread.ManagedThreadId` returns the identifier of the current thread (code file `TaskFoundations/Program.cs`):

```
public static void TraceThreadAndTask(string info)
{
  string taskInfo = Task.CurrentId == null ? "no task" : "task " +
    Task.CurrentId;

  Console.WriteLine($"{info} in thread {Thread.CurrentThread.ManagedThreadId} " +
    $"and {taskInfo}");
}
```

Creating Tasks

Let's start with the synchronous method `Greeting`, which takes a while before returning a string (code file `TaskFoundations/Program.cs`):

```
static string Greeting(string name)
{
  TraceThreadAndTask($"running {nameof(Greeting)}");
  Task.Delay(3000).Wait();
  return $"Hello, {name}";
}
```

To make such a method asynchronously, you define the method `GreetingAsync`. The task-based asynchronous pattern specifies that an asynchronous method is named with the `Async` suffix and returns a `Task`. `GreetingAsync` is defined to have the same input parameters as the `Greeting` method but returns `Task<string>`. `Task<string>` defines a task that returns a `string` in the future. A simple way to return a task is by using the `Task.Run` method. This method creates a new task and starts it. The generic version `Task.Run<string>()` creates a task that returns a `string`. Because the compiler already knows the return type from the implementation (`Greeting` returns a string), you can also simplify the implementation by using `Task.Run()`:

```
static Task<string> GreetingAsync(string name) =>
  Task.Run(() =>
  {
    TraceThreadAndTask($"running {nameof(GreetingAsync)}");
    return Greeting(name);
  });
```

Calling an Asynchronous Method

You can call this asynchronous method `GreetingAsync` by using the `await` keyword on the task that is returned. The `await` keyword requires the method to be declared with the `async` modifier. The code within this method does not continue before the `GreetingAsync` method is completed. However, you can reuse the thread that started the `CallerWithAsync` method. This thread is not blocked (code file `TaskFoundations/Program.cs`):

```
private async static void CallerWithAsync()
{
  TraceThreadAndTask($"started {nameof(CallerWithAsync)}");
  string result = await GreetingAsync("Stephanie");
  Console.WriteLine(result);
  TraceThreadAndTask($"ended {nameof(CallerWithAsync)}");
}
```

When you run the application, you can see from the first output that there's no task. The `GreetingAsync` method is running in a task, and this task is using a different thread from the caller. The synchronous `Greeting` method then runs in this task. As the `Greeting` method returns, the `GreetingAsync` method returns, and the scope is back in the `CallerWithAsync` method after the await. Now, the `CallerWithAsync` method runs in a different thread than before. There's not a task anymore, but although the method started with thread 1, after the `await` thread 4 was used. The `await` made sure that the continuation happens after the task was completed, but it now uses a different thread. This behavior is different between Console applications and applications that have a synchronization context, which is described later in this chapter in the "Async with Windows Apps" section:

```
started CallerWithAsync in thread 1 and no task
running GreetingAsync in thread 4 and task 1
running Greeting in thread 4 and task 1
Hello, Stephanie
ended CallerWithAsync in thread 4 and no task
```

> **NOTE** *The* `async` *modifier can be used with methods that return* `void` *or return an object that offers the* `GetAwaiter` *method. .NET offers the* `Task` *and* `ValueTask` *types. With the Windows Runtime, you also can use* `IAsyncOperation`*. You should avoid using the* `async` *modifier with void methods; read more about this in the "Error Handling" section later in this chapter.*

The next section explains what's driving the `await` keyword. Behind the scenes, continuation tasks are used.

Using the Awaiter

You can use the `async` keyword with any object that offers the `GetAwaiter` method and returns an awaiter. An awaiter implements the interface `INotifyCompletion` with the method `OnCompleted`. This method is invoked when the task is completed. With the following code snippet, instead of using `await` on the task, the `GetAwaiter` method of the task is used. `GetAwaiter` from the `Task` class returns a `TaskAwaiter`. Using the `OnCompleted` method, a local function is assigned that is invoked when the task is completed (code file `TaskFoundations/Program.cs`):

```
private static void CallerWithAwaiter()
{
  TraceThreadAndTask($"starting {nameof(CallerWithAwaiter)}");
  TaskAwaiter<string> awaiter = GreetingAsync("Matthias").GetAwaiter();
  awaiter.OnCompleted(OnCompleteAwaiter);

  void OnCompleteAwaiter()
  {
    Console.WriteLine(awaiter.GetResult());
    TraceThreadAndTask($"ended {nameof(CallerWithAwaiter)}");
  }
}
```

When you run the application, you can see a result similar to the scenario in which you used the `await` keyword:

```
starting CallerWithAwaiter in thread 1 and no task
running GreetingAsync in thread 4 and task 1
running Greeting in thread 4 and task 1
Hello, Matthias
ended CallerWithAwaiter in thread 4 and no task
```

The compiler converts the `await` keyword by putting all the code that follows within the block of an `OnCompleted` method.

Continuation with Tasks

You can also handle continuation by using features of the `Task` object. `GreetingAsync` returns a `Task<string>` object. The `Task` object contains information about the task created and allows waiting for its completion. The `ContinueWith` method of the `Task` class defines the code that should be invoked as soon as the task is finished. The delegate assigned to the `ContinueWith` method receives the completed task with its argument, which allows accessing the result from the task using the `Result` property (code file `TaskFoundations/Program.cs`):

```
private static void CallerWithContinuationTask()
{
  TraceThreadAndTask("started CallerWithContinuationTask");

  var t1 = GreetingAsync("Stephanie");

  t1.ContinueWith(t =>
  {
    string result = t.Result;
    Console.WriteLine(result);

    TraceThreadAndTask("ended CallerWithContinuationTask");
  });
}
```

Synchronization Context

If you verify the thread that is used within the methods, you will find that in all three methods—`CallerWithAsync`, `CallerWithAwaiter`, and `CallerWithContinuationTask`—different threads are used during the lifetime of the methods. One thread is used to invoke the method `GreetingAsync`, and another thread takes action after the `await` keyword or within the code block in the `ContinueWith` method.

With a console application, usually this is not an issue. However, you have to ensure that at least one foreground thread is still running before all background tasks that should be completed are finished. The sample application invokes `Console.ReadLine` to keep the main thread running until the Return key is pressed.

With applications that are bound to a specific thread for some actions (for example, with WPF, UWP, and WinUI applications, UI elements can be accessed only from the UI thread). This is an issue.

Using the `async` and `await` keywords you don't have to do any special actions to access the UI thread after an `await` completion. By default, the generated code switches the thread to the thread that has the synchronization context. A WPF application sets a `DispatcherSynchronizationContext`, and a Windows Forms application sets a `WindowsFormsSynchronizationContext`. Windows apps use the `WinRTSynchronizationContext`. If the calling thread of the asynchronous method is assigned to the synchronization context, then with the continuous execution after the `await`, the same synchronization context is used by default. If the same synchronization context shouldn't be used, you must invoke the `Task` method `ConfigureAwait(continueOnCapturedContext: false)`. An example that illustrates this usefulness is a Windows app in which the code that follows the `await` is not using any UI elements. In this case, it is faster to avoid the switch to the synchronization context.

Using Multiple Asynchronous Methods

Within an asynchronous method, you can call multiple asynchronous methods. How you code this depends on whether the results from one asynchronous method are needed by another.

Calling Asynchronous Methods Sequentially

You can use the `await` keyword to call every asynchronous method. In cases where one method is dependent on the result of another method, this is useful. In the following code snippet, `await` is used with every invocation of `GreetingAsync` (code file `TaskFoundations/Program.cs`):

```
private async static void MultipleAsyncMethods()
{
  string s1 = await GreetingAsync("Stephanie");
  string s2 = await GreetingAsync("Matthias");
  Console.WriteLine($"Finished both methods.{Environment.NewLine} " +
    $"Result 1: {s1}{Environment.NewLine} Result 2: {s2}");
}
```

Using Combinators

If the asynchronous methods are not dependent on each other, it is a lot faster not to `await` on each separately; instead, assign the return of the asynchronous method to a `Task` variable. The `GreetingAsync` method returns `Task<string>`. Both these methods can now run in parallel. Combinators can help with this. A combinator accepts multiple parameters of the same type and returns a value of the same type. The passed parameters are "combined" to one. `Task` combinators accept multiple `Task` objects as parameters and return a `Task`.

The sample code invokes the `Task.WhenAll` combinator method that you can `await` to have both tasks finished (code file `TaskFoundations/Program.cs`):

```
private async static void MultipleAsyncMethodsWithCombinators1()
{
  Task<string> t1 = GreetingAsync("Stephanie");
  Task<string> t2 = GreetingAsync("Matthias");
  await Task.WhenAll(t1, t2);
  Console.WriteLine($"Finished both methods.{Environment.NewLine} " +
    $"Result 1: {t1.Result}{Environment.NewLine} Result 2: {t2.Result}");
}
```

The `Task` class defines the `WhenAll` and `WhenAny` combinators. The `Task` returned from the `WhenAll` method is completed as soon as all tasks passed to the method are completed; the `Task` returned from the `WhenAny` method is completed as soon as one of the tasks passed to the method is completed.

The `WhenAll` method of the `Task` type defines several overloads. If all the tasks return the same type, you can use an array of this type for the result of the `await`. The `GreetingAsync` method returns a `Task<string>`, and awaiting for this method results in a `string`. Therefore, you can use `Task.WhenAll` to return a `string` array:

```
private async static void MultipleAsyncMethodsWithCombinators2()
{
  Task<string> t1 = GreetingAsync("Stephanie");
  Task<string> t2 = GreetingAsync("Matthias");
  string[] result = await Task.WhenAll(t1, t2);
  Console.WriteLine($"Finished both methods.{Environment.NewLine} " +
    $"Result 1: {result[0]}{Environment.NewLine} Result 2: {result[1]}");
}
```

The `WhenAll` method is of practical use when the waiting task can continue only when all tasks it's waiting for are finished. The `WhenAny` method can be used when the calling task can do some work when any task it's waiting for is completed. It can use a result from the task to go on.

Using ValueTasks

Previous to C# 7, the `await` keyword required a `Task` to wait for. Since C# 7, any class implementing the `GetAwaiter` method can be used. A type that can be used with `await` is `ValueTask`. `Task` is a class, but `ValueTask` is a struct. This has a performance advantage because the `ValueTask` doesn't have an object on the heap.

What is the real overhead of a `Task` object compared to the asynchronous method call? A method that needs to be invoked asynchronously typically has a lot more overhead than an object on the heap. Most times, the overhead of a `Task` object on the heap can be ignored—but not always. For example, a method can have one path where data is retrieved from a service with an asynchronous API. With this data retrieval, the data is written to a local cache. When you invoke the method the second time, the data can be retrieved in a fast manner without needing to create a `Task` object.

The sample method `GreetingValueTaskAsync` does exactly this. In case the name is already found in the dictionary, the result is returned as a `ValueTask`. If the name isn't in the dictionary, the `GreetingAsync` method is invoked, which returns a `Task`. This task is awaited. The result received is used to return it in a `ValueTask` (code file `TaskFoundations/Program.cs`):

```
private readonly static Dictionary<string, string> names = new Dictionary<string,
string>();

static async ValueTask<string> GreetingValueTaskAsync(string name)
{
  if (names.TryGetValue(name, out string result))
  {
    return result;
  }
  else
  {
    result = await GreetingAsync(name);
    names.Add(name, result);
    return result;
  }
}
```

The `UseValueTask` method invokes the method `GreetingValueTaskAsync` two times with the same name. The first time, the data is retrieved using the `GreetingAsync` method; the second time, data is found in the dictionary and returned from there:

```
private static async void UseValueTask()
{
  string result = await GreetingValueTaskAsync("Katharina");
  Console.WriteLine(result);
  string result2 = await GreetingValueTaskAsync("Katharina");
  Console.WriteLine(result2);
}
```

If a method doesn't use the `async` modifier and a `ValueTask` needs to be returned, `ValueTask` objects can be created using the constructor passing the result or passing a `Task` object:

```
static ValueTask<string> GreetingValueTask2Async(string name)
{
  if (names.TryGetValue(name, out string result))
  {
    return new ValueTask<string>(result);
  }
  else
  {
    Task<string> t1 =  GreetingAsync(name);
```

```
    TaskAwaiter<string> awaiter = t1.GetAwaiter();
    awaiter.OnCompleted(OnCompletion);
    return new ValueTask<string>(t1);

    void OnCompletion()
    {
      names.Add(name, awaiter.GetResult());
    }
  }
}
```

ERROR HANDLING

Chapter 10, "Errors and Exceptions," provides detailed coverage of errors and exception handling. However, in the context of asynchronous methods, you should be aware of some special handling of errors.

The code for the `ErrorHandling` example makes use of the `System.Threading.Tasks` namespace in addition to the `System` namespace.

Let's start with a simple method that throws an exception after a delay (code file `ErrorHandling/Program.cs`):

```
static async Task ThrowAfter(int ms, string message)
{
  await Task.Delay(ms);
  throw new Exception(message);
}
```

If you call the asynchronous method without awaiting it, you can put the asynchronous method within a `try`/`catch` block—and the exception will not be caught. That's because the method `DontHandle` that's shown in the following code snippet has already completed before the exception from `ThrowAfter` is thrown. You need to await the `ThrowAfter` method, as shown in the example that follows in the next section. Pay attention that the exception is not caught in this code snippet:

```
private static void DontHandle()
{
  try
  {
    ThrowAfter(200, "first");
    // exception is not caught because this method is finished
    // before the exception is thrown
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

> **WARNING** *Asynchronous methods that return* `void` *cannot be awaited. The issue with this is that exceptions that are thrown from* `async void` *methods cannot be caught. That's why it is best to return a* `Task` *type from an asynchronous method. Handler methods or overridden base methods are exempted from this rule because you can't change the return type here. In cases where you need async void methods, it's best to handle exceptions directly within this method; otherwise, the exception can be missed.*

Handling Exceptions with Asynchronous Methods

A good way to deal with exceptions from asynchronous methods is to use `await` and put a `try`/`catch` statement around it, as shown in the following code snippet. The `HandleOnError` method releases the thread after calling the `ThrowAfter` method asynchronously, but it keeps the `Task` referenced to continue as soon as the task is completed. When that happens (which, in this case, is when the exception is thrown after two seconds), the `catch` matches and the code within the `catch` block is invoked (code file `ErrorHandling/Program.cs`):

```
private static async void HandleOnError()
{
  try
  {
    await ThrowAfter(2000, "first");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"handled {ex.Message}");
  }
}
```

Handling Exceptions with Multiple Asynchronous Methods

What if two asynchronous methods are invoked and both throw exceptions? In the following example, first the `ThrowAfter` method is invoked, which throws an exception with the message `first` after two seconds. After this method is completed, the `ThrowAfter` method is invoked, throwing an exception after one second. Because the first call to `ThrowAfter` already throws an exception, the code within the `try` block does not continue to invoke the second method, instead landing within the `catch` block to deal with the first exception (code file `ErrorHandling/Program.cs`):

```
private static async void StartTwoTasks()
{
  try
  {
    await ThrowAfter(2000, "first");
    await ThrowAfter(1000, "second"); // the second call is not invoked
    // because the first method throws
    // an exception
  }
  catch (Exception ex)
  {
    Console.WriteLine($"handled {ex.Message}");
  }
}
```

Now start the two calls to `ThrowAfter` in parallel. The first method throws an exception after two seconds and the second one after one second. With `Task.WhenAll`, you wait until both tasks are completed, whether an exception is thrown or not. Therefore, after a wait of about two seconds, `Task.WhenAll` is completed, and the exception is caught with the `catch` statement. However, you only see the exception information from the first task that is passed to the `WhenAll` method. It's not the task that threw the exception first (which is the second task), but the first task in the list:

```
private async static void StartTwoTasksParallel()
{
  try
  {
```

```
      Task t1 = ThrowAfter(2000, "first");
      Task t2 = ThrowAfter(1000, "second");
      await Task.WhenAll(t1, t2);
    }
    catch (Exception ex)
    {
      // just display the exception information of the first task
      // that is awaited within WhenAll
      Console.WriteLine($"handled {ex.Message}");
    }
  }
```

One way to get the exception information from all tasks is to declare the task variables `t1` and `t2` outside of the `try` block, so they can be accessed from within the `catch` block. Here you can check the status of the task to determine whether they are in a faulted state with the `IsFaulted` property. In case of an exception, the `IsFaulted` property returns true. The exception information itself can be accessed by using `Exception.InnerException` of the `Task` class. Another, and usually better, way to retrieve exception information from all tasks is demonstrated next.

Using AggregateException Information

To get the exception information from all failing tasks, you can write the result from `Task.WhenAll` to a `Task` variable. This task is then awaited until all tasks are completed. Otherwise, the exception would still be missed. As described in the preceding section, with the `catch` statement, only the exception of the first task can be retrieved. However, now you have access to the `Exception` property of the outer task. The `Exception` property is of type `AggregateException`. This exception type defines the property `InnerExceptions` (not only `InnerException`), which contains a list of all the exceptions that have been awaited for. Now you can easily iterate through all the exceptions (code file `ErrorHandling/Program.cs`):

```
private static async void ShowAggregatedException()
{
  Task taskResult = null;
  try
  {
    Task t1 = ThrowAfter(2000, "first");
    Task t2 = ThrowAfter(1000, "second");
    await (taskResult = Task.WhenAll(t1, t2));
  }
  catch (Exception ex)
  {
    Console.WriteLine($"handled {ex.Message}");
    foreach (var ex1 in taskResult.Exception.InnerExceptions)
    {
      Console.WriteLine($"inner exception {ex1.Message}");
    }
  }
}
```

CANCELLATION OF ASYNC METHODS

To cancel asynchronous operations, .NET includes a cancellation framework. The heart of this is the `CancellationToken` that's created from a `CancellationTokenSource` defined in the `System.Threading` namespace. To allow for cleanup of resources, a task should never be killed. To demonstrate how this can be done, the `RunTaskAsync` method receives a `CancellationToken` with a parameter. Within the implementation, the cancellation token is checked if cancellation is requested. If it is, the task has time for cleanup of some resources and exits by invoking the `ThrowIfCancellationRequested` method of the `CancellationToken`. In case cleanup is not required, you can immediately invoke `ThrowIfCancellationRequired`, which throws the `OperationCanceledException` if cancellation is required (code file `TaskCancellation/Program.cs`):

```
Task RunTaskAsync(CancellationToken cancellationToken) =>
  Task.Run(async () =>
  {
    while (true)
    {
      Console.Write(".");
      await Task.Delay(100);
      if (cancellationToken.IsCancellationRequested)
      {
        // do some cleanup
        Console.WriteLine("resource cleanup and good bye!");
        cancellationToken.ThrowIfCancellationRequested();
      }
    }
  });
```

The `Task.Delay` method offers an overload where you can pass the `CancellationToken` as well. This method throws an `OperationCanceledException` as well. If you use this overloaded `Task.Delay` method and need some resource cleanup in the code, you need to catch the `OperationCanceledException` to do the cleanup and re-throw the exception.

When you start the `RunTaskAsync` method, a `CancellationTokenSource` is created. Passing a `TimeSpan` to the constructor cancels the associated token after the specified time. If you have some other task that should do the cancellation, this task can invoke the `Cancel` method of the `CancellationTokenSource`. The `try/catch` block catches the previously mentioned `OperationCanceledException` when cancellation occurs.

```
CancellationTokenSource cancellation = new(TimeSpan.FromSeconds(5));

try
{
  await RunTaskAsync(cancellation.Token);
}
catch (OperationCanceledException ex)
{
  Console.WriteLine(ex.Message);
}
```

ASYNC STREAMS

A great enhancement since C# 8 is the support of async streams. Instead of getting just one result from an asynchronous method, a stream of async results can be received. Async streams is based on the interfaces `IAsyncDisposable`, `IAsyncEnumerable`, and `IAsyncEnumerator`, and updated implementations for the `foreach` and `yield` statements. `IAsyncDisposable` defines the `DisposeAsync` method for asynchronously disposing of resources. `IAsyncEnumerable` corresponds to the synchronous `IEnumerable` interface and defines the `GetAsyncEnumerator` method. `IAsyncEnumerator` corresponds to the synchronous `IEnumerator` interface and defines the `MoveNextAsync` method and the `Current` property. The `foreach` statement has been updated with the syntax `await foreach` to iterate through async streams. The `yield` statement has been modified to support returning `IAsyncEnumerable` and `IAsyncEnumerator`.

> **NOTE** *Read Chapter 6, "Arrays," for information about how the* `foreach` *and* `yield` *statements make use of the synchronous iterator interfaces.*

To see async streams in action, a virtual device represented from the class `ADevice` returns random sensor data in an async stream. The sensor data is defined with the record `SensorData`. The device returns sensor data until it is canceled. Adding the attribute `EnumeratorCancellation` to the `CancellationToken` allows cancellation via an extension method shown later. Within the endless loop implementation, the `yield return` statement is used to return stream values for the `IAsyncEnumerable` interface (code file `AsyncStreams/Program.cs`):

```
public record SensorData(int Value1, int Value2);

public class ADevice
{
  private Random _random = new();
  public async IAsyncEnumerable<SensorData> GetSensorData(
    [EnumeratorCancellation] CancellationToken = default)
  {
    while(true)
    {
      await Task.Delay(250, cancellationToken);
      yield return new SensorData(_random.Next(20), _random.Next(20));
    }
  }
}
```

After defining a method that returns an async stream with the help of the `yield return` statement, let's use this from an `await foreach`. Here, the async stream is iterated, and the cancellation token is passed using the `WithCancellation` method to stop the stream after five seconds:

```
using System;
using System.Threading;
using System.Threading.Tasks;

CancellationTokenSource cancellation = new(TimeSpan.FromSeconds(5));

var aDevice = new ADevice();
try
{
  await foreach (var data in aDevice.GetSensorData().WithCancellation(cancellation.Token))
  {
    Console.WriteLine($"{data.Value1} {data.Value2}");
  }
}
catch (OperationCanceledException ex)
{
  Console.WriteLine(ex.Message);
}
```

> **NOTE** *See Chapter 25, "Services," and Chapter 28, "SignalR," for information about how async streaming can be used to asynchronously stream data across the network.*

ASYNC WITH WINDOWS APPS

Using the `async` keyword with Windows apps works the same as what you've already seen in this chapter. However, you need to be aware that after calling `await` from the UI thread, when the asynchronous method returns, you're back in the UI thread by default. This makes it easy to update UI elements after the asynchronous method is completed.

> **NOTE** *The Windows apps sample code in this chapter uses the new technology WinUI to create a Windows application. Because this technology is so new, please check for updated readme files in the directory of the code samples for what you need to run this application. Using WPF or UWP instead is not a lot different, and you can change the code for these technologies easily.*

Let's create a WinUI Desktop application with Visual Studio. This app contains five buttons and a `TextBlock` element to demonstrate different scenarios (code file `AsyncWindowsApps/MainWindow.xaml`):

```
<StackPanel>
  <Button Content="Start Async" Click="OnStartAsync" Margin="4"/>
  <Button Content="Start Async with ConfigureAwait" Click="OnStartAsyncConfigureAwait"
    Margin="4"/>
  <Button Content="Start Async with Thread Switch"
    Click="OnStartAsyncWithThreadSwitch" Margin="4"/>
  <Button Content="Use IAsyncOperation" Click="OnIAsyncOperation" Margin="4"/>
  <Button Content="Deadlock" Click="OnStartDeadlock" Margin="4"/>
  <TextBlock x:Name="text1" Margin="4"/>
</StackPanel>
```

> **NOTE** *Programming WinUI apps is covered in detail in Chapters 29 through 32.*

In the `OnStartAsync` method, the thread ID of the UI thread is written to the `TextBlock` element. Next, the asynchronous method `Task.Delay`, which does not block the UI thread, is invoked, and after this method is completed, the thread ID is written to the `TextBlock` again (code file `AsyncWindowsDesktopApp/MainWindow.xaml.cs`):

```
private async void OnStartAsync(object sender, RoutedEventArgs e)
{
  text1.Text = $"UI thread: {GetThread()}";
  await Task.Delay(1000);
  text1.Text += $"\n after await: {GetThread()}";
}
```

For accessing the thread ID, WinUI can now use the `Thread` class. With older UWP versions, you need to use `Environment.CurrentManagedThreadId` instead:

```
private string GetThread() => $"thread: {Thread.CurrentThread.ManagedThreadId}";
```

When you run the application, you can see similar output in the text element. Contrary to console applications, with Windows apps defining a synchronization context, after the `await` you can see the same thread as before. This allows direct access to UI elements:

```
UI thread: thread 1
after await: thread 1
```

Configure Await

If you don't need access to UI elements, you can configure `await` not to use the synchronization context. The next code snippet demonstrates the configuration and also shows why you shouldn't access UI elements from a background thread.

With the method `OnStartAsyncConfigureAwait`, after writing the ID of the UI thread to the text information, the local function `AsyncFunction` is invoked. In this local function, the starting thread is written before the asynchronous method `Task.Delay` is invoked. Using the task returned from this method, the `ConfigureAwait` is invoked. With this method, the task is configured by passing the `continueOnCapturedContext` argument set to `false`. With this context configuration, you see that the thread after the await is not the UI thread anymore. Using a different thread to write the result to the `result` variable is okay. What you should never do is shown in the `try` block: accessing UI elements from a non-UI thread. The exception you get contains the `HRESULT` value as shown in the `when` clause. Just this exception is caught in the `catch`: the result is returned to the caller. With the caller, `ConfigureAwait` is invoked as well, but this time the `continueOnCapturedContext` is set to true. Here, both before and after the await, the method is running in the UI thread (code file `AsyncWindowsDesktopApp/MainWindow.xaml.cs`):

```
private async void OnStartAsyncConfigureAwait(object sender, RoutedEventArgs e)
{
  text1.Text = $"UI thread: {GetThread()}";

  string s = await AsyncFunction().ConfigureAwait(
    continueOnCapturedContext: true);

  // after await, with continueOnCapturedContext true we are back in the UI thread
  text1.Text += $"\n{s}\nafter await: {GetThread()}";

  async Task<string> AsyncFunction()
  {
    string result = $"\nasync function: {GetThread()}\n";
    await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
    result += $"\nasync function after await : {GetThread()};";

    try
    {
      text1.Text = "this is a call from the wrong thread";
      return "not reached";
    }
    catch (Exception ex) when (ex.HResult == -2147417842)
    {
      result += $"exception: {ex.Message}";
      return result;
      // we know it's the wrong thread
      // don't access UI elements from the previous try block
    }
  }
}
```

> **NOTE** *Exception handling and filtering is explained in Chapter 10.*

When you run the application, you can see output similar to the following. In the async local function after the `await`, a different thread is used. The text "`not reached`" is never written, because the exception is thrown:

```
UI thread: thread 1
async function: thread 1
async function after await: thread 5; exception: The application called an interface
that was marshalled for a different thread.
after await: thread 1
```

> **NOTE** *In later WinUI chapters in this book, data binding is used instead of directly accessing properties of UI elements. However, with WinUI, you also can't write properties that are bound to UI elements from a non-UI thread.*

Switch to the UI Thread

In some scenarios, there's no effortless way around using a background thread and accessing UI elements. Here, you can switch to the UI thread with the `DispatcherQueue` object that is returned from the `DispatcherQueue` property. The `DispatcherQueue` property is defined in the `DependencyObject` class. `DependencyObject` is a base class of UI elements. Invoking the `TryEnqueu` method of the `DispatcherQueue` object runs the passed lambda expression again in a UI thread (code file `AsyncWindowsDesktopApp/MainWindow.xaml.cs`):

```
private async void OnStartAsyncWithThreadSwitch(object sender, RoutedEventArgs e)
{
  text1.Text = $"UI thread: {GetThread()}";

  string s = await AsyncFunction();

  text1.Text += $"\nafter await: {GetThread()}";

  async Task<string> AsyncFunction()
  {
    string result = $"\nasync function: {GetThread()}\n";
    await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
    result += $"\nasync function after await : {GetThread()}";

    text1.DispatcherQueue.TryEnqueue(() =>
    {
      text1.Text +=
        $"\nasync function switch back to the UI thread: {GetThread()}";
    }
    return result;
  }
}
```

When you run the application, you can see the UI thread used when using `RunAsync`:

```
UI Thread: thread 1
async function switch back to the UI thread: thread 1
async function: thread 1
async function after await: thread 4
after await: thread 1
```

Using IAsyncOperation

Asynchronous methods are defined by the Windows Runtime not to return a `Task` or a `ValueTask`. `Task` and `ValueTask` are not part of the Windows Runtime. Instead, these methods return an object that implements the interface `IAsyncOperation`. `IAsyncOperation` does not define the method `GetAwaiter` as needed by the `await` keyword. However, an `IAsyncOperation` is automatically converted to a `Task` when you use the `await` keyword. You can also use the `AsTask` extension method to convert an `IAsyncOperation` object to a task.

With the example application, in the method `OnIAsyncOperation`, the `ShowAsync` method of the `MessageDialog` is invoked. This method returns an `IAsyncOperation`, and you can simply use the `await` keyword to get the result (code file `AsyncDesktopWindowsApp/MainWindow.xaml.cs`):

```
private async void OnIAsyncOperation(object sender, RoutedEventArgs e)
{
  MessageDialog dlg = new("Select One, Two, Or Three", "Sample");

  dlg.Commands.Add(new UICommand("One", null, 1));
  dlg.Commands.Add(new UICommand("Two", null, 2));
  dlg.Commands.Add(new UICommand("Three", null, 3));

  IUICommand command = await dlg.ShowAsync();

  text1.Text = $"Command {command.Id} with the label {command.Label} invoked";
}
```

Avoid Blocking Scenarios

It's dangerous using `Wait` on a `Task` and the `async` keyword together. With applications using the synchronization context, this can easily result in a deadlock.

In the method `OnStartDeadlock`, the local function `DelayAsync` is invoked. `DelayAsync` waits on the completion of `Task.Delay` before continuing in the foreground thread. However, the caller invokes the `Wait` method on the task returned from `DelayAsync`. The `Wait` method blocks the calling thread until the task is completed. In this case, the `Wait` is invoked from the foreground thread, so the `Wait` blocks the foreground thread. The `await` on `Task.Delay` can never complete, because the foreground thread is not available. This is a classical deadlock scenario (code file `AsyncWindowsDesktopApp/MainWindow.xaml.cs`):

```
private void OnStartDeadlock(object sender, RoutedEventArgs e)
{
  DelayAsync().Wait();
}

private async Task DelayAsync()
{
  await Task.Delay(1000);
}
```

> **WARNING** *Avoid using* `Wait` *and* `await` *together in applications using the synchronization context.*

SUMMARY

This chapter introduced the `async` and `await` keywords. In the examples provided, you've seen the advantages of the task-based asynchronous pattern compared to the asynchronous pattern and the event-based asynchronous pattern available with earlier editions of .NET.

You've also seen how easy it is to create asynchronous methods with the help of the `Task` class and learned how to use the `async` and `await` keywords to wait for these methods without blocking threads. You looked at the error-handling and cancelation aspects of asynchronous methods, and you've seen how async streams are supported with C#. For invoking asynchronous methods in parallel, you've seen the use of `Task.WhenAll`.

For more information on parallel programming and details about threads and tasks, see Chapter 17, "Parallel Programming."

The next chapter continues with core features of C# and .NET and gives detailed information on reflection, metadata, and source generators.